A Real-time Hand Gesture Recognition System

Arun Ganesan
University of Michigan, Ann Arbor

Caoxie (Michael) Zhang
University of Michigan, Ann Arbor

Abstract
Hand gesture recognition can be used in many applications such as interactive data analysis or American Sign Language detection. Current systems are either expensive, unable to run in real time, or require the user wear devices such as custom gloves. We propose an inexpensive solution for predicting hand gestures in real time that uses Microsoft’s Kinect camera. Our system involves training a random forest classifier with a color glove, and then predicting at a pixel level a naked hand. Our system predicts all pixels at about 10 fps, and is resilient to environment differences in prediction. We also conduct extensive experiment studying the random forest classifier and reveal some interesting properties.

1 Introduction
Natural user interfaces (NUI) are a new way for humans to interact with machines. Among numerous NUIs which include multi-touch, eye tracking and motion detection, hand gesture recognition is one promising candidate. In this paper, we design and evaluate a novel hand gesture recognition system to demonstrate that we are close to an actual production-level system. The reader should note that that we do not claim hand gesture recognition is THE future of interfaces. In fact there are some limitations for using hand gestures such as fatigue over long term usage (in the movie Minority Report, Tom Cruise has to take breaks due to fatigue). We focus on the scientific and engineering challenges in building such a system and leave usability studies to future research.

1.1 Design Goals
Our system is designed to maximize user experience. Moreover, our system differs from existing systems in the following ways.

Just hands. Many existing system such as [11, 2] require users to wear gloves or markers to be capture by the camera. We feel this limits the usability of the application and aim to keep the prediction of the system entirely marker free.

Real-time. Our system should run smoothly on average modern computers with a dedicated graphics card. The system should also recognize hand gesture at a high frame rate, nearing real-time speeds. Our desired frame rate 30Hz; we achieved a frame rate around 10Hz. In our design and implementation, one driving goal is to squeeze every milliseconds as possible.

No calibration. Once trained, our system should require no further calibration before working in a new environment.

Robust and accurate. Our system should have an accurate estimation of where the user’s hands are and what gestures they use with low false positive. Moreover, the system should be insensitive to various background, user’s location, camera position and other noise.

Arbitrary gestures. Our system should be able to easily incorporate new types of gestures. With sufficient training, our system should support many complex gestures such as those seen in the American Sign Language.

1.2 Main Ideas
Our system would not be possible without the use of Microsoft Kinect for PC, which we were probably among the first to obtain when it was released in February 2012. Kinect is a multi-purpose sensor including RGB camera, depth camera and audio sensor. The Kinect SDK offers skeletal recognition however lacks the granularity for individual fingers. The SDK also provides raw pixels for the RGB image and depth image at a maximum frame
rate 30Hz. We use the depth image for gesture recognition and both RGB and depth image for generating training samples. The depth image is the key factor that distinguishes our system from most existing systems that use only the RGB camera. The advantage of the depth image is that it offers an additional dimension, i.e. depth of each pixel that is not present in the RGB image. An illustrating example is when an object and its background have a similar color but the depth of the two are drastically different.

Our system adopts a data-driven approach: machine learning as opposed to hand-crafted rule-based systems. The adoption of machine learning transfers the human intelligent efforts from designing rules/algorithms to designing informative features. By extracting the features from labeled data, machine learning algorithms allows computers to learn the rules/algorithms automatically. The key advantages of using machine learning in our system are (1) it is easy to incorporate developer-defined gestures: developers just need to feed the system with the gesture images to be trained rather than deriving new rules to match those gestures and (2) it is robust to various environmental changes such as camera position, background, and various hand sizes: developers just need to generate the gestures on various environments.

In a high-level overview, the system is separated into two parts: training and real-time prediction. Only the real-time prediction component is seen by the end users. In the training component, we use color gloves to generate many labeled data of the depth image. Each pixel in the depth image is labeled as belonging to a gesture or to the background. A random forest classifier is trained to achieve both real-time performance and high accuracy. In the real-time prediction component, the GPU is used to predict the class of each pixel in the depth image and the prediction output is pooled to propose the final position and type of a gesture. Notice that we do not use any temporal or kinetics information as the current simple design suffices for the hand gesture tracking.

1.3 Contributions

We summarize our contributions as follows:

A system for real-time hand gesture recognition. We design and implement a complete real-time hand gesture recognition system based on machine learning. The system broadcasts the location and gesture of the hand through a web socket server and can be used by other applications.

An inexpensive way to generate accurate labeled data. We use color gloves to easily generate labeled data using the aligned RGB and depth images. In [1], the authors use sophisticated computer graphics to generate training samples. We found this expensive and through the use of color gloves, developers can generate their customized gesture without difficulty. Another advantage of our approach is that the system uses actual raw depth images as training samples. These naturally capture realistic noise such as shadows and hardware noise. Using computer generated graphics as done in [1] it is very difficult to simulate these noisy effects. Note that the end users do not need to wear color gloves; they are only used in training.

A computational insight about random forest and support vector machine (SVM). To the best of our knowledge, there seems to be no literature in comparing SVM and random forest from a computational perspective. We provide an in-depth complexity analysis of the two methods rather than merely reporting experimental accuracy as done in most machine learning literature.

Extensive experimental evaluations of the system. We conduct extensive experiments evaluation of the effectiveness of the random forest classifier by systematically exploring a large space of parameters. Interesting results lead to a deeper understanding of random forest.

Our work is made public at: https://github.com/arunganesan/hand-gesture-recognition.

1.4 Outline

The rest of this paper is organized as follows. In section 2 we present an overview of the system architecture. Three large components of the system are described in their own sections. Section 5.1 discusses our choice of random forest over SVM for classification, and the features used in our algorithm. Section 4 explains the collection of training samples, and section 6 discusses the pooling of the predicted pixels. Section 7 discusses some of the details of our implementation. Section 8 presents our experimental findings. Section 9 reflects on our experiences in building the system and working on this project. Section 10 concludes.

2 System Overview

The system architecture is visualized in Figure 1. From the user point of view, our system is divided into two components: developers mode for training gestures and end-user mode for real-time prediction. In a high-level overview, developers using the developer mode supply the system with labeled data with color gloves. The system trains with this labeled data. In the end-user mode,
the camera captures raw depth images of the user’s naked hands. It then predicts every pixel using the GPU and then pools the predicted pixels to get an estimation of the location of the gesture. Between the two modes, they share a component: feature extraction, which obtains the features for each pixel in the depth image. The implementation of feature extraction in the two modes are different - the training component uses the CPU in an off-line fashion and the end-user prediction component uses the GPU (end-user mode) in an on-demand fashion.

2.1 The Kinect Sensor

The Kinect sensor provides both raw depth image and color image at a maximum frame rate of 30Hz and at 640px × 480px resolution. Each pixel in the depth image represent the distance from the object to the camera. The error of the depth can be with several millimeters. However, objects can have a shadow where some parts near the object do not have depth value at all. This is an example of realistic noise that may be missing if the training data was generated using computer graphics methods.

2.2 Pooling & Experience

Experience:

- Adding a virtual wall
- Late optimization
- Test accuracy is not enough

Methodology:

- Mass parallelism: 307,200 threads a frame
- O(\frac{n}{\log n})
- Subject to outliers
- Assumes each cluster has equal size
- Based clustering

3 Feature Extraction and Per-pixel Classification

In this section, we describe the main classification algorithm used in the system. At each frame the system predicts each pixel as belonging to a gesture (e.g., open hand, close hand or background). The prediction result on every pixel are fed into a pooling algorithm to propose the final gesture location and type. The driving reason for us to choose per-pixel classification is that it allows massive parallelism using GPU. The prediction algorithm is identical for every pixel therefore we can leverage the Single Instruction Multiple Data architecture of the GPU.

3.1 Feature Extractions

For each pixel \( x \), we extract a set of features. Each feature corresponds to the depth difference between two offset points, \( \{u,v\} \), from \( x \) and normalized by its depth. This is shown in equation [1] where \( d(x) \) is the depth at point \( x \) and \( \theta \) refers to the pair of offsets \( u, v \).

\[
\theta(x) = d\left(x + \frac{u}{d(x)}\right) - d\left(x + \frac{v}{d(x)}\right) \tag{1}
\]

This feature extraction method is also used in [11]. The offsets \( \theta = \{u,v\} \) are generated randomly. In our design, we randomly sample the offsets from a bounded circular area. \( u \) and \( v \) are obtained as

\[
(r \cos \beta, r \sin \beta), \tag{2}
\]

where \( r \) and \( \beta \) are uniform random variables:

\[
r = U[\min R, \min R] \tag{3}
\]

\[
\beta = U[0, 2\pi] \tag{4}
\]
As we can see from Figure 2, the offset pairs are located within the neighborhood of the palm and are proportional to the depth.

3.2 The Classifier: SVM or Random Forest?

In the training samples each pixel is labeled (we address how to generate massive labeled sample in Section 3.1) and we have to use supervised learning methods. Two methods come into our decision of choice: linear SVM and random forest. We use the following notation in our explanation – the number of training samples: $l$, the number of features: $n$, the number of classes (types of gestures): $c$. Our main criteria for choosing the best algorithm are (1) prediction time and (2) accuracy.

**Linear SVM.** In linear SVM, a model that has $c$ classes and a weight vector of length $n$ is trained. The prediction is based on the dot product between the trained weight vectors and the feature vector. Therefore, the run-time complexity for predicting each pixel is $O(n \times c)$.

**Random Forest.** Random forest is built on an ensemble of decision trees that are trained on bootstrap samples of the training sets. The decision tree in the random forest is a binary tree. Each node in the decision tree has one feature and a threshold to determine which branch to go to by comparing the feature value to the threshold. In the leafs of the tree are labels. The final prediction is made by the majority rule of the all decision trees in the forest. More details about the training and statistical analysis can be seen in [4]. The depth of the decision tree is approximately as $O(\log l)$. Therefore $O(\log l)$ queries of features are made in a single decision tree when predicting. The run-time complexity for making prediction in random forest is then $O(\log l \times n_{\text{tree}})$. In fact, we can prune each decision tree to limit its depth. This refinement, described in Section 3.4, reduces the complexity to $O(d_{\text{tree}} \times n_{\text{tree}})$, for a fixed depth $d_{\text{tree}}$. Finally, we present the random forest prediction algorithm in Algorithm 1.

**Comparison.** Let us do a simple calculation in which we use a realistic parameter setting. Suppose we have 2000 features, 3 classes, and using 3 trees with maximum depth of 20, the random forest is 100 times faster than linear SVM! Moreover, in our experimental evaluation, random forest has proven to be far superior than linear SVM in accuracy. Although linear SVM might use some advanced feature learning technique such as deep learning to achieve similar accuracy as to random forest, SVM is still too slow for us to adopt in the system.

---

1 Bootstrap means uniform sampling with replacement

---

**Algorithm 1 Prediction Algorithm In Random Forest**

Input: a depth image and a pixel $x$ to be predicted

Initialize predict_list

for each tree $i$ in the random forest do

node $\leftarrow$ root of tree $i$

while node is not a leaf do

feature_index $\leftarrow$ node.feature

$(u, v) \leftarrow$ get_offset_pair $(x, \text{feature_index})$

depth $=\text{get_depth_difference}(u, v)$

if depth $>$ node.threshold then

node $\leftarrow$ node’s right children

else

node $\leftarrow$ node’s left children

end if

end while

predict_list[i] $\leftarrow$ node’s label

end for

return predict_list’s majority label

---

Inheriting from decision tree, random forest allows the system to extract features on-demand, which has been extremely crucial for real-time application as in the case of our system. We are really surprised by this analysis since he used to believe linear SVM is unbeatable in practice.

**Training.** Training in linear SVM is very fast as it has a run-time complexity of $O(n \times l)$ and there exists a technique to scale the training to distributed systems [12]. Training in random forest, however, does not have an optimization-based foundation. We use brute force to determine the right feature and right threshold for each node in the decision forest. In determining the right threshold, we use grid search. In the training of random forest, it is highly recommended to put the training data in the main memory.

3.2.1 GPU For Real-time Prediction

There are 307,200 (640 × 480) pixels in a frame. Each pixel will undertake $O(d_{\text{tree}} \times n_{\text{tree}})$ operations for prediction. We first tried to implement the random forest prediction using CPU. It takes about 1 minute to process a frame! GPU presented an attractive alternative as it supports massive parallelism. As can be seen in Figure 3, GPU is more suitable for high-parallelism and high-latency jobs, while CPU is better fit for low-parallelism and low-latency jobs.

In our system, each pixel has fired a thread to make prediction using random forest. In a typical GPU, more than thousands of threads are running concurrently thanks to the special architecture of GPU. As a result, per-pixel classification can be done in about 40ms.
Figure 3: A computational comparison between GPU and CPU. A simple experiment is conducted by varying the number of iterations in a for-loop. GPU would parallelize the for loop.

Figure 4: Two differently colored gloves we used in rapidly generating labeled training samples. For the multicolored glove, we had to ensure the colors were different enough to be easily recognized by the RGB camera.

Figure 5: An example of four gestures. Each gesture represent a label of every pixel that belongs to the glove.

4 Generating training samples

In this section we present our method for rapidly generating labeled training samples.

To rapidly generate training samples, we use a color glove that has a different color for each area of interest. We can present the glove to the camera and extract regions of a specific color using RGB camera and label the corresponding depth pixel appropriately. Examples of gloves we used in our experiments are shown in figure 4.

We devised two different techniques of labeling each pixel. The first technique, shown in the left image of Figure 4, is to differentiate all pixels in the hand from the background. When training the system, we train one gesture at a time and classify a pixel on the hand as belonging to that gesture and classify any pixel not on the hand as part of the background. We trained four gestures using this approach yielding a total of five labels. An example of the gestures can be seen in Figure 5. The second technique is shown in the right image of Figure 4. In this technique each finger is labeled separately. Therefore each pixel can have a total of seven labels - six for the hand, and one for the background.

We also use other techniques to simplify labeling such as cropping and ignoring far away pixels via the depth information.

5 Pooling

After training on the labeled data sets and making per-pixel prediction, our system takes the per-pixel’s predicted class as input and proposes the final location and type of the gesture. We use clustering to achieve this goal, i.e., unsupervised learning. The methodology is as follows: we treat non-background pixels (after per-pixel classification) all equally (without any labels) and then run a clustering algorithm on them. Then we use the largest cluster as a proposal of the hand and use the majority of the predicted label as the type of the gesture. The location of the gesture is found by the 2D median of the largest cluster to add resistance to outliers. We considered several candidate clustering algorithms. First we considered to use mean shift, a similar approach used in [1]. However we found that the complexity of mean shift is very high: $O(N^2)$, where $N$ is the number of pixels (in our case $N = 307200$). We then turned our attention to K-means and DBSCAN clustering.

K-means. K-means is probably the most commonly used clustering algorithm. A parameter K, the number of clusters, has to be specified beforehand. K-means is fast, with a complexity of $O(2N_{\text{non-background}})$, where $N_{\text{non-background}}$ is the number of non-background pixels. We found that however there is a significant weakness in the application of K-means to our system: (1) K-means assumes each cluster has an equal size of points; (2) K-means is not robust to outliers; and (3) K-means requires a fixed number of clusters. As a result, we found that K-means often divides a hand into multiple clusters. Therefore we abandon the use of K-means in our system.

Density-based clustering. Density-based cluster is more suitable for graphics application as it tries to cluster
points that has a density exceeding pre-determine threshold. In addition, density based clustering doesn’t assume a number of clusters ahead of time. We customize the density-based clustering as in Algorithm ?? The complexity is $O(N_{\text{non-background}}\varepsilon^2)$, where $\varepsilon$ is the radius of the interested pixels that is used in Algorithm ??

6 Implementation

There are several implementation details that worth mentioning here.

6.1 Kinect

We purchased two Kinect sensors: one for XBox the other for Windows. Kinect SDK is a Windows-based API allowing us to access the raw depth and RGB images from both versions of Kinect. The Windows version offers a near mode to permit objects to be within 30cm of the sensor. In labeling with the color gloves to generate training samples, we use the built-in function provided by Kinect SDK to map the color pixel to depth pixel. The color detection turns out not to work very well since lighting and camera position introduce many variations to the color. However with cropping and depth-thresholding, we managed to labeled hundreds of images. At this time, we did not use the gloves with fingers colored, as we found (1) it is difficult to produce perfectly built color gloves and (2) the low resolution of the Kinect sensor might not capture the details of figure very well.

6.2 GPU Implementation

From a programmer’s perspective, GPU is seen as an I/O device: one has to move the data from main memory to the GPU memory and move back the processed data to the main memory. There are several libraries for programming on the GPU: Microsoft DirectX, Microsoft DirectCompute, OpenCL, and CUDA. We decided to use OpenCL as it is a standard library for general purpose computing on GPU and can run on almost all GPU (as opposed on CUDA which can only run on NVidia cards). OpenCL uses C as the programming language and has an event-queue framework to process incoming data.

When optimizing the GPU algorithm, we noticed that the cache hit of the model stored in memory was low. We explored reordering the data to increase cache hit rate. Since we store the tree linearly in the memory, we tested (1) using pre-order traversal (depth first), and (2) breadth first traversal for the data structures. It turns out there is no significant difference between the two data structures so we use the first data structure.

6.3 Training Random Forest

The training samples generated is big, usually more than 10 GB. We customized a random forest library [9], for example via changing the double type to float type to save the memory size by half. We realized near the end of the project that optimizing the raining algorithm in addition to the prediction algorithm could have had tremendous gains as training often took longer than 24 hours, with the longest model requiring over 48 hours to train. It is an interesting research topic to parallelize random forest training algorithm to multi-core and distributed systems.

We use Amazon Elastic Computing Cloud (EC2) to offload our massive training workloads to cloud servers.

6.4 Refinements

Several improvements we found are very important for our systems.

Adding a virtual wall. Our system will be placed in a variety of environments and therefore the background will be drastically variable. We decide to add a virtual wall after 1.5m so every pixel farther than 1.5m is treated as 1.5m. We found this technique can deal with a lot of variability of the backgrounds and make prediction more accurate in practice.

6.5 Actual System

Our main application is written in C# in Windows, training algorithm is written in C++, and GPU prediction is written in C. The actual system shows three images: color image, depth image with per-pixel classification, and pooled image with the largest cluster’s position and type. Also the system reports the real-time performance for each stage. An illustrating example is shown in Figure 6.

We built a demo application that maps two gestures onto mouse state. The location of the hand in the XY plane determined the XY location of the mouse; moving the hand to and from the camera moved the mouse wheel up and down; and closing and opening the hand pressed down and released the left mouse button respectively. This simple mapping enabled us to navigate many applications. We successfully demonstrated this mapping on
Figure 6: A snapshot of the actual system. Note the prediction is longer than it is used in a demo machine.

the Google Earth application [10] by panning and zooming the map.

7 Experimental results

In per-pixel classification, there are many parameters involved in capturing the training images, extracting features, and training the random forest classifier. To study the contribution of these parameters, we systematically varied them and studied the accuracy of the resulting classifier. To study the accuracy, we set aside a collection of images as the test set and trained on a different set of images. We didn’t use cross-validation due to the time constraints in training a random forest; it would have taken considerably longer time to test the cross validation accuracy and would have prevented us from running as many tests as we did. We collected images of four different gestures shown in figure 5. We refer to each image that we captured as a “training sample”. Since there are over 300000 pixels in each image, we sampled only 2000 of them so that we can train on more diverse images. From each image, we sampled 1000 background points, and 1000 points from the gesture to ensure we trained evenly on the different types of classes.

7.1 Varying Parameters

We varied three parameters in training the models, such as the number of trees, the number of features, and the number of training samples. We set aside 50 images as the test set, on which we evaluate the trained model to obtain test accuracy.

Number of trees. Figure 7a shows the results of varying the number of trees while fixing the remaining parameters. We observed that the number of trees makes no noticeable difference. In fact, even training the forest with one tree has the same accuracy as with multiple trees. This was also observed for different fixed values of the parameters. Adding multiple trees in the random forest is intended to protect against overfitting. However, our test and training sets were sampled from the same pool of images so we hypothesize one tree is sufficient to model all the variability in the test set. In the actual system, we used 3 trees to make the tradeoff between enough generality of the random forest and efficient performance.

Number of training samples. Figure 7b shows the effect of varying the number of training samples. There is a clear trend of smaller test error as we increase the number of training samples. At 689 training samples, the error is only 1.32%. Increasing the number of training samples had the biggest impact in reducing the error when compared to other factors.

Number of features. Figure 7c shows the effect of varying the number features. Across most of our experiments we observed that using 2000 features outperforms 3000 features, which outperforms 1000 features. However, as seen in figure 7c the difference in error is small. This suggests that if we sampled 1000, 2000, and 3000 features again, we may observe a different trend.

Varying $R_{\max}$ Finally, Figure 7d shows the results of varying $R_{\max}$ in Eq (3), the radius of the offset features. For smaller training sample sizes, setting the radius to 40000 mmpx (short for, mm × pixel, which is the unit of $R_{\max}$) gives the least error classifier. However, as we increase the number of training samples, both 40000 mmpx and 80000 mmpx give similarly accurate classifiers. This means that when we have fewer training samples, setting the radius to 40000 mmpx will allow us to train a more accurate classifier, but when we have enough training samples, the radius is not so important. We notice that making the radius too small or too large gives poor accuracy. By qualitatively checking the offset images as shown in Figure 2, we hypothesize that to train a more accurate classifier, the offsets should cover most of the hand, and then some of the background. If the radius is too small, then the offsets will be right next to each other and feature vectors for different gestures may be too similar.

7.2 Randomness

First, due to the nature of the random forest (because of bootstrapping), the trained model is non-deterministic and may yield a different accuracy rate for the same training set of images. In order to study this randomness, we trained ten models on the same training set with 1000 features, 300 training images, and one tree. We
obtained a mean error of 9.82% with 0.94% standard deviation. Therefore we concluded that the randomness derived from the random forest is negligible.

7.3 SVM Comparison

We compared random forest classifier with a linear SVM classifier, [5]. We trained an SVM classifier for 2000 features, and different number of training images. Figure 7e compares the error rate of a linear SVM with a random forest classifier. Linear SVM only slightly outperforms a random classifier that always predicts a pixel as the background. As we increase the training sample size, linear SVM improves much more slowly than the random forest.

7.4 Pruning

We studied the effect of pruning on the accuracy of prediction. To study this, we pruned the forests trained with 2000 features and three trees for different number of training images. Then we calculated the test set accuracy of the pruned models. It turned out that pruning the tree showed almost no difference in accuracy compared to the full tree. We hypothesize the reason for this is that very high up in the tree the splitting nodes already confidently divide the training points into their class. However, the random forest algorithm continues to exhaust the features until the entire tree is constructed. By stopping the prediction early on in the tree, we do not suffer in accuracy, and improve the speed of prediction.

7.5 Overfitting

Finally, we studied how much the model overfits to the training samples by calculating the training accuracy and comparing that with the test accuracy. We fixed 2000 features and three trees and varied the number of training samples. Figure 7f shows evidence of extreme overfitting. The training error is consistently very low (mean 1.06%) and is much lower than the testing accuracy.

8 Experience

We learned several lessons in building a practical machine learning system.

Late-optimization. Although our system is very time-sensitive, we found that it is not necessary to do optimization on one component while other components are not ready. We found it is best to develop quickly and do profiling to discover the bottleneck and optimize it. This saves us a lot of time in making unnecessary optimization.

Using pipeline rather than events. In the testing of our system, we have incurred many concurrency bugs due to the event-driven programming framework. We solve this by not using locks but adding the processing unit to a pipeline, thus leaving us free concurrency bugs.

Test accuracy is not enough. We found that using the test accuracy is not a good evaluation of the actual system. Sometimes the system performs very well in the test data set, but poorly in practice. The reason behind this is that the environment we evaluate changes all the time: camera position, backgrounds, people’s clothes, and etc. Therefore, we decide to choose some important parameters based on our experience in the actual environment instead of mere test accuracy. For example, although the experiments indicated the number of trees made no difference, we found to this to not be the case in actual tests with varying backgrounds. Therefore, we used a model trained with multiple trees rather than one.

Bundle the model with feature extraction. In our system, we separated feature extraction with the model. This turned out to be error-prone. It would happen sometimes that the trained model has the wrong feature extraction (e.g., the offset pairs are not correct), and things could get even worse when the real-time prediction system mistakenly uses the wrong feature extraction method. Our immediate solution is to be extremely careful to this. However, if we were to build the system again we would have bundled the model with feature extraction to avoid human errors.

8.1 Limitations

Our current system is not without limitations. First, the real-time prediction component cannot achieve a frame rate of 30Hz but only 10Hz on average. Second, the random forest model takes a long time to train, especially with many training samples. Often times, training required around 24 hours or more. Third, the system must be retrained with more training samples every time the user wants to add a new gesture.

9 Related Work

There are two main techniques in hand gesture recognition - appearance based and model based. These are akin to probabilistic models of classification and generative models of classification respectively. Appearance based approaches read the pixels from the camera and build classifiers to label that as belonging to a finite set of classes. The main limitation of this technique is that
the set of labels is finite and fixed ahead of time. The advantage of appearance based approaches is their implementation is often extremely fast and therefore suited for situations where live classification is important. Examples of appearance based techniques can be found in [1, 2]. Model based approaches start with a set of hypotheses of the final classification based on rules of the object being classified. For instance, in the case of hand gestures a hypothesis can be a particular orientation of the joints. An advantage here is that the hypotheses can be generated from an infinite space of possible classifications. The main disadvantage of model-based techniques is that they are often computationally expensive. In addition, model-based approaches tend to be very complex. An example of a model based technique can be found in [3].

As advance sensor technology has become more accessible, many developers and researchers are studying natural interfaces. [7] overviews a variety of input devices for interfacing with 3D models including mouse designs with six degrees of freedom, haptics devices for simulating realistic forces, and computer vision based techniques for head and hand tracking. Very relevant to our project, [8] presents a method for identifying 25 gestures in 3D using the gyroscope and accelerometer in the Nintendo Wii remote. This falls in a more general category called “spatially convenient input-devices”.

The work most similar to ours is from Microsoft [1] on the details of the Kinect’s appearance-based skeleton recognition algorithm. They classify each pixel as belonging to some portion of the body and then pool all pixels from each portion to determine the joint positions. Their training set is generated from synthetic 3D images of body orientations. The feature extraction technique and the random forest classification algorithm used in this work is borrowed from [4]. Our project differs from this in the method for generating training samples, and the scale of classification.

[2] present another appearance-based method for detecting hand gestures using just an RGB camera. Their solution requires the user wear a glove with a special pattern imprinted on it. The camera pictures the hand in different orientations, normalizes the image, and identifies the nearest neighbor in a database of common hand gestures. Unlike this approach, our technique only uses a color glove for training purposes.

[3] present a model-based approach using the Kinect. They first generate a set of hypotheses with knowledge of inverse kinematics, and approximate shape of fingers and hands. They evaluate these hypotheses and pick the most likely one based on the depth image from the Kinect. This technique is able to detect hand gestures even in the place of occlusions and is implemented efficiently reaching up to 15 updates per second. However, being model-based, this technique may be vulnerable to differences in hand shapes, and still lags behind in speed compared to
10 Conclusion

We presented a real-time hand gesture recognition system built on top of Microsoft’s Kinect SDK using their depth range camera. We are able to achieve real-time prediction at about 10 fps due to the efficient use of random forest using GPU. We extensively studied the random forest classifier by varying the multitude of hyperparameters. We discovered that (1) increasing the number of training samples had the biggest impact in the test set accuracy, (2) changing the number of trees had no noticeable effect, and (3) random forest classifier significantly outperforms a linear SVM. We built a demo application that maps two gestures to mouse position, click status, and wheel movement. We then used this mapping in the Google Earth application and were able to navigate with pan and zoom using just our gestures.

We decided to make our implementation open source in hopes of attracting other developers to test and develop our application. We found this to be lacking with many other research projects in this area. They report the results, but do not make the code and testing data available.

Our experiments were done on four gestures, and the demo was with two gestures. We would like to train our system with more gestures and more diverse human hands to discover limitations of our system. For instance, such a system will be useful in detecting American Sign Language alphabet gestures.

<table>
<thead>
<tr>
<th>Algorithm 2 Density-based clustering</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> a depth image with labeled pixels</td>
</tr>
<tr>
<td><strong>for</strong> each non-background and unvisited pixel <em>p</em> <strong>do</strong></td>
</tr>
<tr>
<td>mark <em>p</em> as visited</td>
</tr>
<tr>
<td>neighbor_list ← get_neighbors(<em>p, ε</em>)</td>
</tr>
<tr>
<td><strong>if</strong> len(neighbor_list) &gt; ε² × density <strong>then</strong></td>
</tr>
<tr>
<td>add <em>p</em> to a new cluster</td>
</tr>
<tr>
<td>create a queue: Seed</td>
</tr>
<tr>
<td>enqueue neighbor_list to Seed</td>
</tr>
<tr>
<td><strong>while</strong> Seed not empty <strong>do</strong></td>
</tr>
<tr>
<td>pixel <em>t</em> ← Seed.Dequeue()</td>
</tr>
<tr>
<td>neighbor_list ← get_neighbors(<em>t, ε</em>)</td>
</tr>
<tr>
<td><strong>if</strong> len(neighbor_list) &gt; ε² × density <strong>then</strong></td>
</tr>
<tr>
<td>explore_neighbors(neighbor_list)</td>
</tr>
<tr>
<td><strong>end if</strong></td>
</tr>
<tr>
<td><strong>end while</strong></td>
</tr>
<tr>
<td><strong>end if</strong></td>
</tr>
<tr>
<td><strong>end for</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>function GET_NEIGHBORS(<em>q, ε</em>)</th>
</tr>
</thead>
<tbody>
<tr>
<td>list is empty</td>
</tr>
<tr>
<td><strong>for</strong> each pixel <em>x</em> that is within <em>ε</em> distance of <em>q</em> <strong>do</strong></td>
</tr>
<tr>
<td><strong>if</strong> <em>x</em> is labeled as non-background <strong>then</strong></td>
</tr>
<tr>
<td>add <em>x</em> into list</td>
</tr>
<tr>
<td><strong>end if</strong></td>
</tr>
<tr>
<td><strong>end for</strong></td>
</tr>
<tr>
<td>return list</td>
</tr>
<tr>
<td><strong>end function</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>function EXPLORE_NEIGHBORS(neighbor_list)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>for</strong> each pixel <em>q</em> in neighbor_list <strong>do</strong></td>
</tr>
<tr>
<td><strong>if</strong> <em>q</em> is not visited <strong>then</strong></td>
</tr>
<tr>
<td>Seed.Enqueue(<em>q</em>)</td>
</tr>
<tr>
<td>mark <em>q</em> as visited</td>
</tr>
<tr>
<td>add <em>q</em> to the new cluster</td>
</tr>
<tr>
<td><strong>end if</strong></td>
</tr>
<tr>
<td><strong>end for</strong></td>
</tr>
<tr>
<td>return neighbor_list</td>
</tr>
<tr>
<td><strong>end function</strong></td>
</tr>
</tbody>
</table>
References


